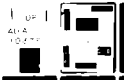
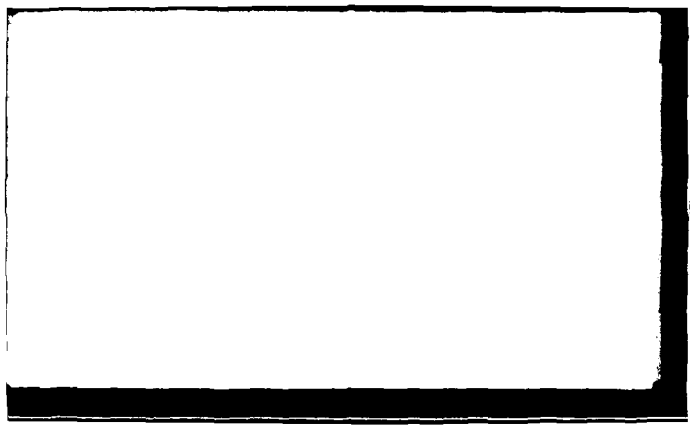
AD-A103 751 RENSSELAER POLYTECHNIC INST TROY NY DEPT OF MATHEMAT--ETC F/G 9/2
A LAMBDA-CALCULUS MODEL FOR GENERATING VERIFICATION CONDITIONS.(U)
JUN 81 S K ABDALI, F WINKLER N00014-75-C-1026
UNCLASSIFIED RPI-CS-8104 NL



END
DATE FILMED
10-81
DTIC



(14) AFI-

Technical Report CS-8104

(6) A LAMBDA-CALCULUS MODEL FOR GENERATING VERIFICATION CONDITIONS.

(10) S. Kamal Abdali
Franz Winkler

(11) June 1981

(12) 32

(9) Technical rept.

Prepared for

U.S. Office of Naval Research
Contract Number N00014-75-C-1026
(15)

Mathematical Sciences Department
Rensselaer Polytechnic Institute
Troy, New York 12181

Introduction

The most well-known program verification technique is based upon the Floyd-Naur idea of inductive assertions [4]: A programming language command imposes certain fixed implications between the relations holding among the values of program variables just before and just after the execution of that command. The (partial) correctness of a program can thus be proved if the output specification claimed at the program exit are derived from the input specifications assumed at the entrance by following the chain of implications mentioned above for all entrance-to-exit control flow paths in the program. Usually, this requires 1) the invention of a number of assertions associated with some key points ("cutpoints") in the program 2) the generation of the implications mentioned above ("verification conditions") for every pair of adjacent points chosen, and 3) the demonstration (possibly, using the services of a theorem-prover) that each of these implications is true. Of these, the second task -- the generation of verification conditions is strictly a mechanical process requiring substitution and simple arithmetical evaluation. The lambda-calculus has built-in rules to carry out the process of substitution and can be easily augmented with arithmetical evaluation rules. Thus, it seems reasonable to seek a lambda-calculus-based method for the automatic generation of verification conditions.

In this paper we develop such a method. It has been obtained by extending an existing [1] lambda-calculus model of programming languages in which programs are translated into lambda-expressions such that the (numerical) execution of programs is modelled by the lambda-calculus process of reduction. In the new model, a program is effectively translated into a lambda-expression whose reduction yields a list of all verification conditions. The extension from the previous to the new model is non-trivial, for we are now interested in a sense in the symbolic, rather than numerical, evaluation of programs.

For generating verification conditions, one must have a program as well as inductive assertions associated with certain

properly chosen cutpoints in the program. We specify a programming language in which inductive assertions are incorporated within the program body by means of special assert statements. Equipped with assignments, conditionals, compounds, ALGOL-type blocks, and loops, this language is simple yet quite powerful. We then present a set of translation rules mapping the statements of the specified programming language into lambda-expressions. Using these rules, a program can be effectively translated into a lambda-expression, say by extending the compiler of [5]. Finally, we show that the model is correct in the sense that the translation of any program produced by our rules does indeed give all verification conditions.

Verification Conditions

Given a program, in the flowchart form, say, and the program input and output conditions, the inductive assertion method to prove the partial correctness of the program proceeds as follows ([4], explanation in [7]). First, cutpoints are chosen on the flowchart edges such that there is at least one cutpoint in each loop. Cutpoints are also placed on the start and halt edges. Next, to each cutpoint is associated a predicate -- the inductive assertion -- which is intended to express the relation holding among the values of the program variables each time the control passes that cutpoint. The desired input and output conditions of the program serve as the assertions at the start and halt cutpoints, respectively. Next, a verification condition is constructed for each basic path -- a path which begins and ends at two (not necessarily different) cutpoints but does not pass through any other cutpoint. The verification condition for a basic path α from cutpoint i to cutpoint j states that if the assertion at i is true and the control traverses α, then the assertion at j will hold (with the new values of variables attained at j). Finally, each verification is proved to be true. By induction it is then the case that the assertion at each cutpoint is true whenever control reaches that cutpoint (assuming that the input condition on the start edge is satisfied at the initiation of program execution). In particular, the assertion at the halt edge is true whenever control reaches this edge, that is, whenever the program halts. Thus the program is partially correct with respect to the given input and output conditions.

In constructing the verification condition for a given path, one has to take into account the transformation in variable values resulting from the execution of the statements in the path. For example, let a path consist of a single assignment statement $x:=x+1$ and let the assertions at the beginning and the end of the path be $x^2+2x+3>0$ and $x^2+2>0$, respectively. The verification condition should be equivalent to the statement: If x^2+2x+3 is true for some value of x, and the assignment $x:=x+1$ is executed, then $x^2+2>0$ is true for the new value of x. Clearly this is not

equivalent to

$$x^2+2x+3>0 \supset x^2+2>0,$$

for the predicates $x^2+2x+3>0$ and $x^2+2>0$ hold for different values of x, namely those respectively before and after the execution of x:=x+1. We can "normalize" the predicates so as to make them refer to the same values of x, either before or after the execution of the assignment statement. In terms of the values existing before the execution, the predicates are $x^2+2x+3>0$ and $(x+1)^2+2>0$; in terms of the values after the execution, they are $(x-1)^2+2(x-1)+3>0$ and $x^2+2>0$. The verification condition can then be written in the equivalent forms

$$x^2+2x+3>0 \supset (x+1)^2+2>0$$

or, $$(x-1)^2+2(x-1)+3>0 \supset x^2+2>0.$$

In general, suppose the assertions at the beginning and the end of a path α are P and Q, respectively. Then the verification condition for the path is a predicate $P'\&R \supset Q'$, where R represents the condition under which α is traversed (R=true, if α does not contain any conditional statement), and P', Q' are obtained from P, Q, respectively, by making appropriate substitutions to reflect the changes in variable values effected by the execution of statements in α. The substitutions should be done so as to make P' and Q' refer to the same values of variables. (R should be derived to also correspond to the same values of variables.) In the special case that the predicates are to be expressed in terms of the variable values at the beginning of the path, P' is just P, and Q' is formed from Q by "backward substitution" [6]: The path is traced backward and for every assignment statement encountered, the assigned expression is substituted for the assigned variable; the cumulative effect of all such substitutions is to transform Q into Q'. On the other hand, if the predicates are to be expressed in terms of the variable values at the end of the path, then Q' is just Q, and P' is obtained from P by an analogous process of "forward substitution."

Since the lambda-calculus ([3,9]) contains built-in rules to carry out the process of substitution, it is possible to use a lambda-calculus-based method for the automatic generation of

verification conditions. The method to be described in this paper has been obtained by modifying and extending the lambda-calculus model of programming languages described in [1]. This model is comprised of rules for translating programs written in a large subset of ALGOL 60 (or a similar language) into lambda-expressions in such a manner that if the result of executing a program P with inputs $i_1,\ldots,i_m$ consists of outputs $o_1,\ldots,o_n$, then the lambda-expression $(\{P\}\{i_1\}\ldots\{i_m\})$ reduces to the tuple or list $\langle\{o_1\},\ldots,\{o_n\}\rangle$. (Here, {...} denotes the lambda-calculus representation of the enclosed object.) Based upon these rules, a compiler has been constructed ([5]) to translate PASCAL programs into lambda-expressions. The goal of the model to be presented in this paper is to provide rules for translating any program suitably annotated with assertions into a lambda-expression whose reduction yields a list of the lambda-calculus representations of the verification conditions. To distinguish the two models, we call the former the "execution model" and the latter the "verification model".

The Source Language

Before giving any translation rules for the verification model, we must specify the language, call it PL, in which the programs acceptable by the model can be written. This language contains the following features:

1. Integer and boolean data types
2. The usual arithmetical, boolean, and relational operators
3. Assignment statements of the form variable := expression
4. Input and output statements of the form
   read variable list
   write expression list
5. Conditional statements of the form
   if condition then statement
   if condition then statement else statement
6. Compound statements and blocks as in ALGOL 60.

Inductive assertions associated at chosen cutpoints in a flowchart are incorporated directly in the body of a PL program by means of the following statements:

7. Assert statement of the form
   assert assertion
8. Maintain-while statement of the form
   Maintain assertion while condition do statement

The features (1) to (6) have the usual ALGOL 60 semantics. The effect of the execution of an assert statement is the following: The assertion is evaluated. If it is true, then control passes to the next statement; if false, an error exception occurs. The effect of the execution of a maintain-while statement is the following: The assertion is evaluated. If it is true, then the while-do part is executed according to the usual semantics; if false, an error exception occurs.

A variable occurring in any statement (3) through (8) (as a left-hand part or an operand in any condition or expression) must be a variable in whose scope the statement occurs, that is, must be a variable in the "environment" of the statement (see [1]). However, a variable occurring in an assertion in any statement (7) or (8) may be a variable of the environment of the statement

or it may be one of the special variables $o, i_1, \ldots, i_m$ where m is fixed for each program. Of these variables, o is called the output variable, and i_j are called input variables. The need for these variables will be clear later.

PL, the source language for our verification model, is much simpler than the source languages used in the execution models of [1,5], yet it contains more features than in [6,7], say. The verification model can be easily extended to include in PL such features as multiple assignments of ALGOL 60, collateral (parallel) assignments of ALGOL 68, for and repeat statements of PASCAL, array data type, and functions without side-effects. But the incorporation of general procedures seems difficult.

The Verification Model: Preliminaries

In the execution model [1,5], the lambda-expression representation of each statement (of ALGOL or PASCAL, say) has been derived using the following idea: Each statement in a program may be thought of as manipulating 1) the variables accessible at the time the statement is executed (these constitute the statement's "environment"), and 2) an entity identifying the point in the program that is being executed. This entity, called the "continuation" or "program remainder", is nothing but an eventually recursive description of the entire portion of the program not executed so far. The statement can therefore be translated as an abstraction with respect to the continuation (denoted by the variable ϕ) and the indeterminates representing the program variables. Referring the reader to [1,5] for the actual details of representation, we give below some examples of translation in the execution model.

Example. Some translations in the execution model

Environment: (x,y,x)

Statement	Representation
y:=x+3;	$a \equiv \lambda\phi xyz:\phi x(+x3)z$
if y=1 then z:=0 else x:=z+1;	$b \equiv \lambda\phi xyz:(=y1)cd\phi xyz$, where $c \equiv \lambda\phi xyz:\phi xy0$ $d \equiv \lambda\phi xyz:\phi(+z1)yz$
while y>x do x:=x+z;	$e \equiv \lambda\phi xyz:(>yz)(f\phi)\phi xyz$ $f \equiv \lambda\phi xyz:\phi(+xz)yz$
write x+3;	$g \equiv \lambda\phi xyzo:\phi xyzo;(+x3)$
read x,z;	$g \equiv \lambda\phi xyzoi_1i_2:\phi i_1yi_2o$

The representation of variables, constants, operations, relations, and expressions in the verification model is the same as in [1,5]. But when translating statements, we need some other constituents besides the continuation and the environment. These are:

Variable Stacks. There is a fundamental difference between the

execution of a conditional statement for a numerical result and the symbolic evaluation for generating verification conditions. Whenever a conditional statement is reached during a numerical execution, some condition is evaluated and according to the result of the evaluation, the first or the second branch of the statement is taken. In the verification context, however, we actually have to execute both branches of a conditional statement, and moreover it is essential to start the computation of each branch with the same values of the various program variables. We solve that problem by keeping a stack for every variable. Each time we encounter a conditional statement, the current values of the program variables are pushed on the stack, and when we pass the corresponding ELSE these values are retrieved.

Assertions. Essentially what we have to do in order to generate the verification conditions for a program is to traverse every basic path of the program between inductive assertions and output the lemma:
"assertion at start point & path condition ⊃ assertion at end point with appropriately changed values of variables".
So we need some constituent which allows us to store the assertion of the start point and successively add the path condition. This leads us to the concept of an assertion constituent.

Verification conditions. Whenever a verification condition is generated we want to store it in some constituent which finally will be the output of the whole process.

Thus the translation of a statement into the verfication model if of the following form:

$$\lambda\ \phi\ \tau\ \alpha\ v_1\ \sigma_1 \ldots v_n\ \sigma_n\ o\ i_1 \ldots i_m : \phi'\tau'\alpha'v_1'\sigma_1'\ldots v_n'\sigma_n'o'$$

program remainder
assertions
verification conditions
program variables
variable stacks
output
inputs
new values to reflect the effect of the statement

Following are some definitions and abbreviations that will be used later:

$I \equiv \lambda x{:}x$ (Identity, null list or triple)

$\Omega \equiv (\lambda xy{:}xx)(\lambda xy{:}xx)$ (Undefined value)

$\langle a_1,\ldots,a_n\rangle \equiv \lambda x{:}xa_1\ldots a_n,\ n \geq 0$ (list or triple)

$s_{11} \equiv \lambda x{:}xI$ (Note: $s_{11}\langle a\rangle \rightarrow a$)

$s_{21} \equiv \lambda x{:}x(\lambda xy{:}x)$ (Note: $s_{21}\langle a,b\rangle \rightarrow a$)

$s_{22} \equiv \lambda x{:}x(\lambda xy{:}y)$ (Note: $s_{22}\langle a,b\rangle \rightarrow b$)

$a;b \equiv \lambda x{:}axb$ (Note: $\langle a_1,\ldots,a_n\rangle;b \rightarrow \langle a_1,\ldots,a_n,b\rangle$)

$\underline{push} \equiv \lambda xy{:}\langle x,y\rangle$ (Note: $\underline{push}\ ab \rightarrow \langle a,b\rangle \leftarrow a;b$

$\underline{push}\ \langle a_1,\ldots,a_n\rangle b \rightarrow \langle a_1,\ldots,a_n,b\rangle$)

$\underline{pop} \equiv s_{22}$ (Note: $pop\ \langle a_1,\ldots,a_n,b\rangle \rightarrow b$)

$\underline{add} \equiv \lambda xy{:}\underline{push}((s_{21}y)\&x)(s_{22}y)$

(Note: $\underline{add}\ a\langle b,c\rangle \rightarrow \langle b\&a,c\rangle$)

$\underline{ch} \equiv \lambda x{:}\underline{push}(s_{21}(s_{22}x))(\underline{push}(s_{21}x)(s_{22}(s_{22}x)))$

(Note: $\underline{ch}\langle a,\langle b,c\rangle\rangle \rightarrow \langle b,\langle a,c\rangle\rangle$)

$\underline{comb} \equiv \lambda x{:}\underline{push}((s_{21}x)\vee(s_{21}(s_{22}x)))(s_{22}(s_{22}x))$

(Note: $\underline{comb}\langle a,\langle b,c\rangle\rangle \rightarrow \langle avb,c\rangle$)

Translation Rules

Using the notation of [1] (to which the reader is referred for motivation and explanation), we now list the translation rules of the verification model. These rules have the form

$\{S\}_E \equiv$ the lambda-expression representing statement S in environment E.

Assignment statement

$$\{v_i := e\}_{(v_1,\ldots,v_n)} \quad (e \text{ is an expression})$$

$$\equiv \lambda\phi\tau\alpha v_1\sigma_1\ldots v_n\sigma_n : \phi\tau\alpha v_1\sigma_1\ldots v_{i-1}\sigma_{i-1}\{e\}\sigma_i\ldots v_n\sigma_n$$

Input-Output statement

$$\{\underline{\text{read}}\ v_i\}_{(v_1,\ldots,v_n)}$$

$$\equiv \lambda\phi\tau\alpha v_1\sigma_1\ldots v_n\sigma_n o x : \phi\tau\alpha v_1\sigma_1\ldots v_{i-1}\sigma_{i-1}x\sigma_i v_{i+1}\sigma_{i+1}\ldots v_n\sigma_n o$$

$$\{\underline{\text{write}}\ e\}_{(v_1,\ldots,v_n)}$$

$$\equiv \lambda\phi\tau\alpha v_1\sigma_1\ldots v_n\sigma_n o : \phi\tau\alpha v_1\sigma_1\ldots v_n\sigma_n o;\{e\}$$

Compound statement

$$\{\underline{\text{begin}}\ S_1;S_2;\ldots;S_n\ \underline{\text{end}}\}_{(v_1,\ldots,v_n)}$$

$$\equiv \lambda\phi:\ \{S_1\}(\{S_2\}(\ldots(\{S_n\}\phi)\ldots))$$

Blocks

$$\{\underline{\text{begin}}\ \langle\text{type}\rangle u_1;\ldots\langle\text{type}\rangle u_m;S_1;S_2;\ldots;S_p\ \underline{\text{end}}\}_{(v_1\ldots v_n)}$$

$$\equiv \lambda\phi\tau\alpha:\{S_1\}_F(\{S_2\}_F(\ldots(\{S_p\}_F(\lambda\tau\alpha u_1\sigma_{u_1}\ldots u_m\sigma_{u_m}:\phi\tau\alpha))\ldots))\tau\alpha\underbrace{\Omega I\Omega I\ldots\Omega I}_{m \text{ times}}$$

where $F = (u_1,\ldots,u_m,v_1,\ldots,v_m)$ = the environment extended by the newly declared variables of the block.
Since in the verification model the current assertion α and the list of verification conditions τ precede the representation of the variables, we have to include α and τ in the specification of a block.

For every new variable, which is introduced by the block, we need a stack. This stack is initially empty (I) and has to be deleted at the end of the block together with the variable.

Conditional Statements

$$\{\underline{\text{if}}\ b\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\}_{(v_1,\ldots,v_n)}$$

$\equiv \lambda\phi:\underline{as}(\{S_1\}(\underline{su}(\{S_2\}(\underline{sc}\ \phi))))$.

Subsidiary definitions:

$\underline{as} \equiv \lambda\phi\tau\alpha v_1\sigma_1 \ldots v_n\sigma_n:\phi\tau(\underline{push}(s_{21}\alpha\ \&\ b)(\underline{add}(\tilde{}b)\alpha)$
$v_1(\underline{push}\ v_1\sigma_1)\ldots v_n(\underline{push}\ v_n\sigma_n)$

This takes the first part of the assertion (which represents the valid assertion at the point of the condition statement) $s_{21}\alpha$ and creates two versions of it, which in addition to $s_{21}\alpha$ also assume b or $\tilde{}b$, respectively. Furthermore, the current values of the variables are pushed onto their stacks. This is necessary, because now we want to perform the statement $\{S_1\}$, which might change the values of the variables. But we need these values for the execution of $\{S_2\}$ later on.

$\underline{su} \equiv \lambda\phi\tau\alpha v_1\sigma_1 \ldots v_n\sigma_n:\phi\tau(\underline{ch}(\underline{add}(\bar{v}_1=v_1\ \&\ \ldots\ \&\ \bar{v}_n=v_n)\alpha))$
$(s_{21}\sigma_1)\ (\underline{pop}\ \sigma_1)\ \ldots\ (s_{21}\sigma_n)\ (\underline{pop}\ \sigma_n)$

This saves the results of the execution of $\{S_1\}$ by adding them to the current assumption. Now we are ready to switch the first two branches of the "assumption tree", which causes the version containing $\tilde{}b$ to become the current assumption. For the following execution of $\{S_2\}$ we delete the action of $\{S_1\}$ on the variables and restore their old values, which is achieved by unstacking them.

$\underline{sc} \equiv \lambda\phi\tau\alpha v_1\sigma_1 \ldots v_n\sigma_n:\phi\tau(\underline{comb}(\underline{add}(\bar{v}_1=v_1\ \&\ \ldots\ \&\ \bar{v}_n=v_n)\alpha))$
$\bar{v}_1\sigma_1\ldots\bar{v}_n\sigma_n$

This finally saves the results of the execution of $\{S_2\}$ by adding them to the current assumption. Afterwards the assumptions for the then and else clauses are combined to a single one by disjunction. Since the results of both $\{S_1\}$ and $\{S_2\}$ are now saved in the current assertion and are denoted by v_i, we make $\bar{v}_i$ the new value of v_i.

Note: The second form of IF-statement

$\{\underline{if}\ b\ \underline{then}\ S\}_{(v_1,\ldots,v_n)}$

is translated into

$\lambda\phi:\underline{as}(\{S_1\}(\underline{su}(I(\underline{sc}\ \phi))))$

Assert statements

{assert $a(v_1,\ldots,v_n,o)$}

$\equiv \lambda\phi\tau\alpha v'_1\sigma_1\ldots v'_n\sigma_n o' : \phi\tau;(s_{21}\alpha \supset a')$ (sub a α)
$v_1\sigma_1\ldots v_n\sigma_n o$

where

$a' \equiv a(v'_1,\ldots,v'_n,o')$

sub $\equiv \lambda xy$:push x $(s_{22}y)$

The lemma: "current assertion => a with the variables replaced by their current values" is added to the verification conditions. Afterwards the current assumption is replaced by the assumption a and we delete the former values of the variables.

Maintain-while statement

{maintain a while b do S}

$\equiv \lambda\phi\tau\alpha v_1\sigma_1\ldots v_n\sigma_n$:ast($ad_1$({S}(ast($ad_2\phi$)))) $\tau\alpha v_1\sigma_1\ldots v_n\sigma_n$

where

ast $\equiv \lambda\phi\tau\alpha v'_1\sigma_1\ldots v'_n\sigma_n o' : \phi\tau;(s_{21}\alpha \supset a')$ (sub a α)
$v_1\sigma_1\ldots v_n\sigma_n o$

is a representation of the statement

assert a,

$ad_1 \equiv \lambda\phi\tau\alpha v_1\sigma_1\ldots v_n\sigma_n : \phi\tau$(add b α) $v_1\sigma_1\ldots v_n\sigma_n$

adds the predicate b to the current assumption.

Now the statement {S} is performed and afterwards "ast" checks whether the predicate a has been maintained. This procedure sets up the necessary verification conditions for the maintain-while statement. What remains to te done is to add ~b to the current assumption and examine the program remainder:

$ad_2 \equiv \lambda\phi\tau\alpha v_1\sigma_1\ldots v_n\sigma_n : \phi\tau$(add(~b)$\alpha$) $v_1\sigma_1\ldots v_n\sigma_n$.

Examples of Translations from PL into the Lambda-calculus

We now present some examples of translations of programs obtained by using the rules presented above. The program statements have been tagged with identifiers used as the names of the corresponding lambda-expressions. In writing lambda-expressions, certain notational liberties have been taken in order to make them human-readable; the intended correct form must be obvious in these cases. Thus, expressions have been written in the usual infix notation, rather than the proper postfix lambda-expressions. For example,

[gcd(n,m) = gcd(x,y) & $x \geq 0$ & $y \geq 0$]

has been used as a shorthand for

(&(& (= (gcd n m)(gcd x y))($\geq$ x 0))($\geq$ y 0)).

The result of reductions given at the end of each example has been obtained by means of a computer program [2] which reduces lambda-expressions to their simplest (normal) forms. The actual computer print-out is included with one example.

Example 1. Summation of a given number of consecutive integers.

Input condition: $n \geq 0$ (n is input)

Output condition: output = $\frac{n(n+1)}{2}$

```
begin integer m,s;                          .............................p
   read m;(*The value n is assigned to variable m*).....re
   s:=0;                                    .............................a1
   begin integer j;                         .............................bl1
      j:=1;                                 .............................a2
      maintain                              .............................mw
         s=(j-1)*j/2 and m=n and j≤m+1.....................ast
         while j≤m do                       .............................adl,ad2
            begin                           .............................bl2
               s:=s+j;                      .............................a3
               j:=j+1                       .............................a4
            end
   end;
   write s; (* s is appended to the (currently empty) output..wr
               file o.  Thus, s11o=s *)
```

assert $s_{11}o = m*(m+1)/2$ .....ter

end

Final Result.....res

Translations

re ≡ $\lambda\phi\tau\alpha\ m\sigma_m s\sigma_s o i_1 : \phi\tau\alpha i_1 \sigma_m s\sigma_s o$

a1 ≡ $\lambda\phi\tau\alpha\ m\sigma_m s\sigma_s : \phi\tau\alpha m\sigma_m 0\sigma_s$

a2 ≡ $\lambda\phi\tau\alpha j : \phi\tau\alpha 1$

ast ≡ $\lambda\phi\tau\alpha j'\sigma_j m'\sigma_m s'\sigma_s : \phi(\tau;(s_{21}\alpha \supset [s'=(j'-1)*j'/2 \ \&\ m'=n' \ \&\ j' \leq m+1]))$
$(\underline{sub}\ [s=(j-1)*j/2 \ \&\ m=n \ \&\ j \leq m+1]\alpha)\ j\sigma_j m\sigma_m s\sigma_s$

ad1 ≡ $\lambda\phi\tau\alpha : \phi\tau(\underline{add}\ [j \leq m]\alpha)$

ad2 ≡ $\lambda\phi\tau\alpha : \phi\tau(\underline{add}\ [j > m]\alpha)$

a3 ≡ $\lambda\phi\tau\alpha j\sigma_j m\sigma_m s\sigma_s : \phi\tau\alpha j\sigma_j m\sigma_m [s+j]\sigma_s$

a4 ≡ $\lambda\phi\tau\alpha j\sigma_j : \phi\tau\alpha [j+1]\sigma_j$

bℓ2 ≡ $\lambda\phi$:a3(a4 ϕ)

mw ≡ $\lambda\phi$:ast(ad1(bl2(ad2 ϕ))))

bℓ1 ≡ $\lambda\phi\tau\alpha$:a2(mw($\lambda\tau\alpha j\sigma_j : \phi\tau\alpha$))$\tau\alpha\Omega$I

wr ≡ $\lambda\phi\tau\alpha m\sigma_m s\sigma_s o : \phi\tau\alpha m\sigma_m s\sigma_s (o;s)$

ter ≡ $\lambda\phi\tau\alpha m'\sigma_m s'\sigma_s o' : \phi(\tau;(s_{21}\alpha \supset [s_{11}o'=m'*(m'+1)/2]))$
$(\underline{sub}\ [s_{11}o=m*(m+1)/2]\alpha)\ m\sigma_m s\sigma_s o$

p ≡ $\lambda\phi\tau\alpha$:re(al(bℓ1(wr(ter($\lambda\tau\alpha m\sigma_m s\sigma_s : \phi\tau\alpha$)))))$\tau\alpha\Omega$I$\Omega$I

res ≡ p($\lambda\tau\alpha o : \tau$) I <n≥0,I> In

By lambda-calculus reduction, we obtain:

res → $\langle [n \geq 0 \supset 0=0 \ \&\ n=n \ \&\ 1 \leq n+1],\ [s=\frac{i(i-1)}{2} \ \&\ \ell=n \ \&\ i \leq \ell+1 \ \&\ i \leq \ell$

$\supset\ s+i = \frac{(i+1)(i+1-1)}{2} \ \&\ \ell=n \ \&\ i+1=\ell+1],\ [s=\frac{i(i-1)}{2} \ \&\ \ell=n$

$\&\ i \leq \ell+1 \ \&\ i>\ell \supset s=\frac{\ell(\ell+1)}{2}\ ]\rangle$

Thus, res is a list containing the three required verification conditions. It is easily seen that each of these conditions is true. So the program is partially correct with respect to the specified input and output conditions.

Example 2. Square-root program.

Input condition: $n \geq 0$ (n is input)

Output condition: output = $\max_{k \geq 0} k^2 \geq n$

```
begin integer x,y1,y2,y3;                          .....p
   begin                                           .....aa
      read x;  (* The value n is assigned to variable x *)...re
      (y1,y2,y3):=(0,0,1);                         .....a1
      y2:=y2+y3                                    .....a2
   end;
   maintain                                        .....mw
      x=n and y1^2≤n and y2=(y1+1)^2 and y3=2*y1+1 .....ast
      while y2≤x do                                .....ad1,ad2
         (y1,y2,y3):=(y1+1,y2+y3+2,y3+2);          .....bb
   write y1;                                       .....wr
    (* y1 is appended to the (currently empty) file o.  Thus,
       s11o=y1 *)
   assert (s11o)^2≤n and n<((s11o)+1)^2            .....ter
end
                                                   Final Result.....res
```

Translations

$re \equiv \lambda\phi\tau\alpha x\sigma_x y_1\sigma_{y_1} y_2\sigma_{y_2} y_3\sigma_{y_3} o i_1 : \phi\tau\alpha i_1\sigma_x y_1\sigma_{y_1} y_2\sigma_{y_2} y_3\sigma_{y_3}$

$a1 \equiv \lambda\phi\tau\alpha x\sigma_x y_1\sigma_{y_1} y_2\sigma_{y_2} y_3\sigma_{y_3} : \phi\tau\alpha x\sigma_x 0\sigma_{y_1} 0\sigma_{y_2} 1\sigma_{y_3}$

$a2 \equiv \lambda\phi\tau\alpha x\sigma_x y_1\sigma_{y_1} y_2\sigma_{y_2} y_3\sigma_{y_3} : \phi\tau\alpha x\sigma_x y_1\sigma_{y_1} [y_2+y_3]\sigma_{y_2} y_3\sigma_{y_3}$

$aa \equiv \lambda\phi : re(a1(a2\ \phi))$

$ast \equiv \lambda\phi\tau\alpha x'\sigma_x y_1'\sigma_{y_1} y_2'\sigma_{y_2} y_3'\sigma_{y_3} : \phi(\tau; (s_{21}\alpha \supset [x'=n\ \&\ y_1'^2 \leq n\ \&\ y_2'=(y_1'+1)^2\ \&\ y_3'=2*y_1'+1]))$

$(\underline{sub}\ [x=n\ \&\ y_1^2 \leq n\ \&\ y_2=(y_1+1)^2\ \&\ y_3=2*y_1+1]\alpha) x\sigma_x y_1\sigma_{y_1} y_2\sigma_{y_2} y_3\sigma_{y_3}$

$ad1 \equiv \lambda\phi\tau\alpha : \phi\tau(\underline{add}\ [y_2 \leq x]\alpha)$

$ad2 \equiv \lambda\phi\tau\alpha : \phi\tau(\underline{add}\ [y_2 > x]\alpha)$

$bb \equiv \lambda\phi\tau\alpha x\sigma_x y_1\sigma_{y_1} y_2\sigma_{y_2} y_3\sigma_{y_3} : \phi\tau\alpha x\sigma_x [y_1+1]\sigma_{y_1} [y_2+y_3+2]\sigma_{y_2} [y_3+2]\sigma_{y_3}$

$mw \equiv \lambda\phi{:}ast(adl(bb(ast(ad2\ \phi))))$

$wr \equiv \lambda\phi\tau\alpha x\sigma_x y_1\sigma_{y_1} y_2\sigma_{y_2} y_3\sigma_{y_3} o{:}\phi\tau\alpha x\sigma_x y_1\sigma_{y_1} y_2\sigma_{y_2} y_3\sigma_{y_3}(o;y_1)$

$ter \equiv \lambda\phi\tau\alpha x'\sigma_x y_1'\sigma_{y_1} y_2'\sigma_{y_2} y_3'\sigma_{y_3} o'{:}\phi(\tau;(s_{21}\alpha \supset [(s_{11}o')^2 \leq n \ \&\ n<((s_{11}o')+1)^2]))$

$(\underline{sub}\ [(s_{11}o)^2 \leq n \ \&\ n<((s_{11}o)+1)^2]\alpha)\ x\sigma_x y_1\sigma_{y_1} y_2\sigma_{y_2} y_3\sigma_{y_3} o$

$p \equiv \lambda\phi\tau\alpha{:}aa(mw(wr(ter(\lambda\tau\alpha x\sigma_x y_1\sigma_{y_1} y_2\sigma_{y_2} y_3\sigma_{y_3}{:}\phi\tau\alpha))))\tau\alpha\Omega I\Omega I\Omega I\Omega I$

$res \equiv p(\lambda\tau\alpha o{:}\tau)\ I\ <n\geq 0,\ I>\ In$

By lambda-calculus reduction, we obtain:

$res \rightarrow <[n\geq 0 \supset 0\leq n \ \&\ 1=1 \ \&\ 1=1 \ \&\ n=n],$

$[y_1^2 \leq n \ \&\ y_2=(y_1+1)^2 \ \&\ y_3=2y_1+1 \ \&\ x=n \ \&\ y_2 \leq x$

$\supset (y_1+1)^2 \leq n \ \&\ y_2+y_3+2=(y_1+1+1)^2 \ \&\ y_3+2=2(y_1+1)+1 \ \&\ x=n],$

$[y_1^2 \leq n \ \&\ y_2=(y_1+1)^2 \ \&\ y_3=2y_1+1 \ \&\ x=n \ \&\ y_2>x$

$\supset y_1^2 \leq n \ \&\ n<(y_1+1)^2]>$

Thus, res is a list containing the three required verification conditions. It is easily seen that each of these conditions is true. So the program is partially correct with respect to the specified input and output conditions.

Example 3. GCD calculation.

Input condition: $n\geq 0$ & $m\geq 0$ (m,n are inputs).

Output condition: output = gcd(n,m)

```
begin integer x,y;                          .............................h
   read x,y; (* Input values n,m assigned to x,y *).........a
   maintain
      gcd(n,m)=gcd(x,y) & x≥0 & y≥0.........................ast
      while x≠y do                          .............................adl,ad2
         if x>y then                        .............................as,sc,d,e
            x:=x-y                          .............................b
         else                               .............................su
            y:=y-x;                         .............................c
   write x;                                 .............................f
     (* Value of x appended to (currently empty) output
        file o.  Thus s11o=x *)
```

assert $\gcd(n,m)=s_{11}o$ .....g

end

Final Result.....res

Translations

$a \equiv \lambda\phi\tau\alpha\ x\sigma_x y\sigma_y o i_1 i_2 : \phi\tau\alpha i_1\sigma_x i_2\sigma_y o$

$ast \equiv \lambda\phi\tau\alpha x'\sigma_x y'\sigma_y : \phi(\tau;(s_{21}\alpha \supset [\gcd(n,m)=\gcd(x',y') \ \&\ x'\geq 0 \ \&\ y'\geq 0]))$
$(\underline{sub}\ [\gcd(n,m)=\gcd(x,y) \ \&\ x\geq 0 \ \&\ y\geq 0]\alpha)x\sigma_x y\sigma_y$

$adl \equiv \lambda\phi\tau\alpha x\sigma_x y\sigma_y : \phi\tau(\underline{add}[x\neq y]\alpha)x\sigma_x y\sigma_y$

$ad2 \equiv \lambda\phi\tau\alpha x\sigma_x y\sigma_y : \phi\tau(\underline{add}[x=y]\alpha)x\sigma_x y\sigma_y$

$as \equiv \lambda\phi\tau\alpha x\sigma_x y\sigma_y : \phi\tau(\underline{push}(s_{21}\alpha \ \&[x>y])(\underline{add}[x\leq y]\alpha))x(\underline{push}\ x\sigma_x)y(\underline{push}\ y\sigma_y)$

$b \equiv \lambda\phi\tau\alpha x\sigma_x y\sigma_y : \phi\tau\alpha[x-y]\sigma_x y\sigma_y$

$su \equiv \lambda\phi\tau\alpha x\sigma_x y\sigma_y : \phi\tau\underline{ch}(\underline{add}[\bar{x}=x \ \&\ \bar{y}=y]\alpha)(s_{21}\sigma_x)(\underline{pop}\ \sigma_x)(s_{21}\sigma_y)(\underline{pop}\ \sigma_y)$

$c \equiv \lambda\phi\tau\alpha x\sigma_x y\sigma_y : \phi\tau\alpha x\sigma_x[y-x]\sigma_y$

$sc \equiv \lambda\phi\tau\alpha x\sigma_x y\sigma_y : \phi\tau\underline{comb}(\underline{add}[\bar{x}=x \ \&\ \bar{y}=y]\alpha)\bar{x}\sigma_x\bar{y}\sigma_y$

$d \equiv \lambda\phi : as(b(su(c(sc\ \phi))))$

$e \equiv \lambda\phi : ast(adl(d(ast(ad2\ \phi))))$

$f \equiv \lambda\phi\tau\alpha x\sigma_x y\sigma_y o : \phi\tau\alpha x\sigma_x y\sigma_y(o;x)$

$g \equiv \lambda\phi\tau\alpha x'\sigma_x y'\sigma_y o' : \phi(\tau;(s_{21}\alpha \supset [\gcd(n,m)=s_{11}o']))$
$(\underline{sub}[\gcd(n,m)=s_{11}o]\alpha)x\sigma_x y\sigma_y o$

$h \equiv \lambda\phi\tau\alpha : a(e(f(g(\lambda\tau\alpha\ x\sigma_x y\sigma_y : \phi\tau\alpha))))\tau\alpha\Omega I\Omega I$

$res \equiv h(\lambda\tau\alpha o : \tau)I<[n\geq 0 \ \&\ m\geq 0],\ I>\ I\ n\ m$

By lambda-calculus reduction, we obtain

$res \rightarrow <[n\geq 0 \ \&\ m\geq 0 \supset \gcd(n,m)=\gcd(n,m) \ \&\ n\geq 0 \ \&\ m\geq 0],$
$[\gcd(n,m)=\gcd(x,y) \ \&\ x\geq 0 \ \&\ y\geq 0 \ \&\ x\neq y \ \&\ x\leq y \ \&\ \bar{x}=x \ \&\ \bar{y}=y-x$
$\underline{or}\ \gcd(n,m)=\gcd(x,y) \ \&\ x\geq 0 \ \&\ y\geq 0 \ \&\ x\neq y \ \&\ x>y \ \&\ \bar{x}=x-y \ \&\ \bar{y}=y$
$\supset \gcd(n,m)=\gcd(\bar{x},\bar{y}) \ \&\ \bar{x}\geq 0 \ \&\ \bar{y}\geq 0],$
$[\gcd(n,m)=\gcd(x,y) \ \&\ x\geq 0 \ \&\ y\geq 0 \ \&\ x=y \supset \gcd(n,m)=x]>$

Thus, res is a list containing the required verification conditions. As each verification condition is true, the program is partially correct with respect to the specified input and output conditions.

Translations of individual statements into the lambda-calculus.
(Note: L stands for λ on computer input.)

```
A =   (L PHI  (L TAU(L AL(L VX(L SX(L VY  (L SY(L O1(L I1(L I2
      (PHI TAU AL I1 SX I2 SY O1)))))))))))).

AST =(L PHI(L TAU(L AL  (L VXP(L SX(L VYP(L SY(PHI  (PSH TAU
      ((S21 AL)IMP (EQU (GCD N M)  (GCD VXP VYP) AND (GE VXP 0)
     AND (GE VYP 0))))
      (SUB (EQU (GCD N M)  (GCD VX VY) AND (GE VX 0) AND
      (GE VY 0)) AL) VX SX VY SY)))))))).

AD1 =(L PHI(L TAU(L AL(L VX(L SX(L VY  (L SY  (PHI TAU  (ADD
      (NE VX VY) AL)  VX SX VY SY)))))))).

AD2 =(L PHI(L TAU(L AL(L VX(L SX(L VY  (L SY  (PHI TAU  (ADD
      (EQU VX VY) AL)  VX SX VY SY)))))))).

AS =  (L PHI(L TAU(L AL(L VX  (L SX(L VY(L SY(PHI TAU  (PSH
      ((S21 AL)  AND  (GT VX VY))  (ADD (LE VX VY) AL))  VX
      (PSH VX SX)  VY  (PSH VY SY)))))))))).

BB =  (L PHI(L TAU(L AL(L VX  (L SX(L VY(L SY  (PHI TAU  AL
      (- VX VY)  SX VY SY  )))))))).

SU =  (L PHI(L TAU(L AL(L VX(L SX(L VY(L SY  (PHI TAU  (CH
      ( ADD  ((EQU VXB VX)  AND  (EQU VYB VY))  AL))  (S21 SX)
      (S22 SX)  (S21 SY)  (S22 SY)))))))))).

CC =  (L PHI(L TAU(L AL  (L VX(L SX  (L VY(L SY  (PHI TAU  AL
      VX SX  (-VY VX)  SY)))))))).

SC =  (L PHI(L TAU  (L AL(L VX(L SX(L VY  (L SY  (PHI TAU  (COM
      (ADD  ((EQU VXB VX)  AND  (EQU VYB VY))  AL))
      VXB SX VYB SY)))))))).

D =   (L PHI  (AS(BB(SU(CC(SC PHI))))).

E =   (L PHI  (AST(AD1(D(AST(AD2 PHI))))).

F =   (L PHI(L TAU(L AL(L VX(L SX  (L VY(L SY  (L O1  (PHI TAU  AL
      VX SX VY SY  (PSH O1 VX)))))))))).

G =   (L PHI(L TAU(L AL  (L VXP(L SX(L VYP(L SY  (L OP  (PHI
      (PSH TAU  ((S21 AL)  IMP  (EQU  (GCD N M)  (S11 OP))))
      (SUB  (EQU  (GCD N M)  (S11 O1))  AL)  VX SX VY SY C1)))))))))).

H =   (L PHI(L TAU(L AL  ((A  (E  (F  (G  (L TAU  (L AL(L VX  (L SX(L VY
      (L SY  (PHI TAU AL)))))))))))  TAU AL CM I CM I)))).

RES =H(L TAU(L AL(L OUT TAU)))I(PSH(GE N 0 AND GE M 0)I)I N &.
```

Definition of auxiliary objects

```
PSH =(L A(L B(L X (X A B)))).

SEC =(L X(L Y Y)).

S11 =T I.      (Alternative definition— T ≡ λXY:yx is a primitive)

S21 =T K.

S22 =T SEC.

SUB =(L VX(L VY (PSH VX (S22 VY)))).

ADD =(L VZ(L X (PSH ((S21 X) AND VZ) (S22 X)))).

CH = (L VZ (PSH (S21 (S22 VZ)) (PSH (S21 VZ) (S22 (S22 VZ))))).

COM =(L VZ ((PSH (OR (S21 VZ) (S21 (S22 VZ)))) (S22 (S22 VZ)))).
```

Now RES should reduce to a list of three verification conditions.
Due to the definition of lists, RES has the form <<<P>,Q>,R>.
The components are extracted below in the order R, Q, and P.

```
   S22(S21 (S21 RES)).

INPUT OBJECT IS...
  : S22(S21(S21 RES))

REDUCED OBJECT IS...
    GE N 0 AND GE M 0
    IMP  (EQU (GCD N M) (GCD N M) AND (GE N 0) AND (GE M 0))


   S22(S21 RES).

INPUT OBJECT IS...
  : S22(S21 RES)

REDUCED OBJECT IS...
    OR  (EQU (GCD N M) (GCD VX VY) AND (GE VX 0)
         AND (GE VY 0) AND (NE VX VY) AND (LE VX VY)
         AND (EQU VXB VX AND (EQU VYB(-VY VX))))
        (EQU (GCD N M) (GCD VX VY) AND (GE VX 0)
         AND (GE VY 0) AND (NE VX VY) AND (GT VX VY)
         AND (EQU VXE(- VX VY) AND (EQU VYB VY)))
    IMP (EQU (GCD N M) (GCD VXB VYB) AND (GE VXE 0)
         AND(GE VYE 0))
```

```
    S22 RES.

INPUT OBJECT IS...
  : S22 RES


REDUCED OBJECT IS...
    EQU (GCD N M) (GCD VX VY) AND (GE VX 0) AND (GE VY 0)
    AND (EQU VX VY)
    IMP (EQU (GCD N M) VX)
```

The Correctness of Verification Model

We would now like to prove that the verification model presented above is correct. In other words, we would like to show that the translation of a PL program according to the above-given rules indeed reduces to a list containing the lambda-calculus representation of all verification conditions. We begin with some definitions:

A path α in a PL program is said to be basic if it
--starts with an "assert" or "maintain" or starts at the beginning of the program,
--ends with an "assert" or "maintain", and
--does not contain any other "assert" or "maintain".

The verification condition for a basic path α with starting assertion q, terminating assertion p and path condition γ (the condition under which γ is traversed) is

$$q \ \& \ \gamma \Rightarrow p,$$

where the variables in q are replaced by the result of performing α.

The verification condition list for a PL-program is a list of verification conditions of all its basic paths.

The predicate after an "assert" or "maintain" is referred to as an assertion.

With these definitions the following theorem holds.

Theorem: If the assert (and maintain) statements in a PL-program P contain only program variables, symbols $i_1,\ldots,i_n$ for the input, the symbol o for the output and constants, and {prog} is a translation of P into the verification model, and α_0 is the input assertion, then

$$\{prog\}\ (\lambda\tau\alpha o{:}\tau)\ I\ \langle\alpha_0, I\rangle\ I\ i_1,\ldots,i_n$$

generates the verification condition list for P.

Proof:
$(\lambda\tau\alpha o{:}\tau)$ as the final program remainder deletes all the information but τ, the verification condition list.
We show that for all basic paths leading from assertions $q_1,\ldots,q_k$ (*) via path conditions $\gamma_1,\ldots,\gamma_n$ to the assertion p in a PL-program,

(*) Some of the q_j's may be equal.

the corresponding translation into the verification model generates the verification conditions

$$q_1 \ \& \ \gamma_1 \Rightarrow p(\bar{x})$$

and

$$\vdots$$

and

$$q_k \ \& \ \gamma_k \Rightarrow p(\bar{x})$$

and adds them to τ, the list of verification conditions. $\bar{x}$ denotes the values of the variables immediately before the assertion, possibly also containing the value of the output variable o.

Proof by structural induction:

basis:

q and p follow each other immediately, after execution of assert q, α_1 is q.

The path cond. $\gamma \equiv T$.

The variables hold $x_1,\ldots,x_n$.

assert p generates the lemma

$$s_{21}{}^{\alpha} \Rightarrow p|_{x=x}$$

which is true.

induction step:

(a) Suppose the hypothesis holds for arbitrary $p,q_1,\ldots,q_k$ in a PL-program. Now consider a PL-program where

$x_i := e(x_1,\ldots,x_n);$
assert r

is substituted for

assert p.

Let α denote the stack of assertions before the execution of the assignment, which is the same as the one in the original program before assert p.

Let $\bar{x}$ be the variable values before the assignment.

If we now let $p(x) \equiv r(x_1,\ldots,x_{i-1},e(x_1,\ldots,x_n),x_{i+1},\ldots,x_n)$ we know from the hypothesis

$$s_{21}{}^{\alpha} \Rightarrow p|_{x=\bar{x}}$$

is equivalent to

$$\bigwedge_{j=1,\ldots,k} (q_j \;\&\; \gamma_j \Rightarrow p|_{x=\bar{x}}$$

and furthermore to

$$\bigwedge_{j=i,\ldots,k} (q_j \;\&\; \gamma_r \Rightarrow r|_{x=(\bar{x}_1,\ldots,e(\bar{x}_1,\ldots,\bar{x}_n),\ldots\bar{x}_n))}$$

(b) Suppose the hypothesis holds for arbitrary $p,q_1,\ldots,q_k$ in a PL-program. Now consider a program where

begin <type> $u_1,\ldots,$<type> u_m;
assert $r(u_1,\ldots,u_m,x_1,\ldots,x_n)$

is substituted for

assert p.

Let α denote the stack of assertions before the execution of the assignment, which is the same as the one in the original program before assert p.

Let $\bar{x}$ be the variable values before the begin.

If we now let

$$p(x) = r(\underbrace{\Omega,\ldots,\Omega}_{m},x_1,\ldots,x_n)$$

we know from the hypothesis

$$s_{21}\alpha \Rightarrow p|_{x=\bar{x}}$$

is equivalent to

$$\bigwedge_{j=1,\ldots,k} (q_j \;\&\; \gamma_j \Rightarrow p|_{x=\bar{x}})$$

and furthermore to

$$\bigwedge_{j=1,\ldots,k} (q_j \;\&\; \gamma_j \Rightarrow r|_{u=(\Omega),x=\bar{x}})$$

(c) Suppose the hypothesis holds for arbitrary $p,q_1,\ldots,q_k$ in a PL-program. Now consider the program where

end; (<type>$u_1,\ldots,$<type>u_m)
assert $r(x_1,\ldots,x_n)$

is substituted for

assert p.

Let α... stack of assertions before end

$\bar{x}$, $\bar{u}$... variable values before end.

(1) u_i did not overwrite some global variable x_j. Then if we let

$p(u_1,\ldots,u_m,x_1,\ldots,x_n) := r(x_1,\ldots,x_n)$

we know from the hypothesis:

$$s_{21}\alpha \Rightarrow p|_{u=\bar{u},x=\bar{x}}$$

is equivalent to

$$\bigwedge_{j=1,\ldots,k} (q_j \;\&\; \gamma_j \Rightarrow p|_{x=\bar{x}})$$

and futhermore to

$$\bigwedge_{j=1,\ldots,k} (q_j \;\&\; \gamma_j \Rightarrow r|_{x=\bar{x}})$$

(2) If however u_i overwrote the global variable x_j, we could change the name of u_i, so that this phenomenon does not occur and we get

$$\bigwedge_{j=1,\ldots,k} (q_j \;\&\; \gamma_j \Rightarrow r|_{x=\bar{x}})$$

(d) Suppose the hypothesis holds for arbitrary $p,q_1,\ldots,q_k$ in a PL-program. Now consider the program where

```
if b(x1,...,xn) then
    assert r
  :
```

is substituted for

```
assert p(x1,...,xn).
```

Let α... stack of assertions before *if*

$\bar{x}$... variable values before *if*

The execution of the *if* changes

$\alpha = \langle\alpha_1,\ldots\rangle$ to $\langle\alpha_1 \;\&\; b(\bar{x}),\langle a_i \;\&\; \tilde{}b(\bar{x}),\ldots\rangle\rangle$

and leaves $\bar{x}$ unchanged.

If we let

$$p(x) = (b(x) \Rightarrow r(x))$$

then we know from the hypothesis that

$$s_{21}\alpha \Rightarrow p(\bar{x})$$

generates the correct verification conditions

$$\bigwedge_{j=1,\ldots,k} (q_j \;\&\; \gamma_j \Rightarrow p(\bar{x})).$$

So

$$s_{21}\alpha \;\&\; b(\bar{x}) \Rightarrow r(\bar{x})$$

which is equivalent to

$$s_{21}\alpha \Rightarrow \underbrace{(b(\bar{x}) \Rightarrow r(\bar{x}))}_{p(\bar{x})}$$

Also generates

$$\bigwedge_{j=1,\ldots,k} (q_j \;\&\; \gamma_j \Rightarrow p(\bar{x})) \quad \langle\!=\!=\!\rangle \quad \bigwedge_{j=1,\ldots,k} (q_j \;\&\; \gamma_j \;\&\; b(\bar{x}) \Rightarrow r(\bar{x}))$$

which are the correct verification conditions for the paths leading to assert r in the modified program.

(e) Suppose the hypothesis holds for arbitrary $p, q_1, \ldots, q_n$ in a PL-program. Now consider the program where

```
if b(x) then
   :
else
   assert r(x)
```

is substituted for

```
assert p.
```

Let α... stack of assertions before if, x... variable values before if
The execution of if changes

$\alpha = \langle\alpha_1,\ldots\rangle$ to $\langle\alpha_1 \;\&\; b(\bar{x}), \langle\alpha_1 \;\&\; \tilde{}b(\bar{x}),\ldots\rangle\rangle$

and puts the values $\bar{x}$ on the variable stacks.
else causes

$\langle\alpha_1', \langle a_1 \;\&\; \tilde{}b(\bar{x}),\ldots\rangle\rangle$ to be changed to $\langle\alpha_1 \;\&\; \tilde{}b(\bar{x}), \langle\alpha_1',\ldots\rangle\rangle$

and the variable values are retrieved from the stacks. A reasoning analogous to that in (d) now causes $s_{21}\alpha \;\&\; \tilde{}b(\bar{x}) \Rightarrow r(\bar{x})$ to be equivalent to the correct verification conditions

$$\bigwedge_{j=1,\ldots,k} (q_j \;\&\; \gamma_j \;\&\; \tilde{}b(\bar{x}) \Rightarrow r(\bar{x})).$$

(f) Suppose the hypothesis holds for arbitrary

$p^t, q_1^t, \ldots, q_k^t, p^e, q_1^e, \ldots, q_L^e$ in a PL-program

```
   :
if b(x) then
   begin
      :
      assert p^t(x)
   end
else
   begin
      :
      assert p^e(x)
   end;
```

Now consider the program where assert p^t, p^e are eliminated and replaced by assert r(x) after the if-statement.

```
if b(x) then
   begin
   .
   end
else
   begin
     :
   end;
assert r(x)
```

Let α... stack of assertions before p^t

α'... stack of assertions before p^e

α''... stack of assertions after if-statement

$\bar{x}^t$... variable values before p^t

$\bar{x}^e$... variable values before p^e.

From the induction hypothesis we know

$$s_{21}\alpha \Rightarrow p^t(\bar{x}^t)$$

is equivalent to

$$\bigwedge_{j=1,\ldots,k} (q_j^t \;\&\; \gamma_j^t \Rightarrow p^t(\bar{x}^t))$$

and

$$s_{21}\alpha' \Rightarrow p^e(\bar{x}^e)$$

is equivalent to

$$\bigwedge_{j=1,\ldots,L} (q_j^e \;\&\; \gamma_j^e \Rightarrow p^e(\bar{x}^e)).$$

$\alpha' = \langle s_{21}\alpha', \langle s_{21}\alpha \;\&\; \tilde{x}=\bar{x}^t, \ldots\rangle\rangle$

$\alpha'' = \langle (s_{21}\alpha \;\&\; \tilde{x}=\bar{x}^t) \vee (s_{21}\alpha' \;\&\; \tilde{x}=\bar{x}^e), \ldots\rangle$

the variable values after the if-statement are $\tilde{x}$.

Let $p^t(x) \equiv (\tilde{x}=x \Rightarrow r(\tilde{x}))$

$p^e(x) \equiv (\tilde{x}=x \Rightarrow r(\tilde{x}))$

So

$$s_{21}\alpha'' \Rightarrow r(\tilde{x})$$

is equivalent to

$$(s_{21}\alpha \;\&\; \tilde{x}=\bar{x}^t \Rightarrow r(\tilde{x})) \;\&\; (s_{21}\alpha' \;\&\; \tilde{x}=\bar{x}^e \Rightarrow r(\tilde{x}))$$

or

$$(s_{21}\alpha \Rightarrow p^t(\bar{x}^t)) \;\&\; (s_{21}\alpha' \Rightarrow p^e(\bar{x}^e))$$

$\langle\Rightarrow\rangle$

$$\bigwedge_{j=1,\ldots,k} (q_j^t \;\&\; \gamma_j^t \Rightarrow r(\bar{x}^t)) \;\&\; \bigwedge_{j=1,\ldots,L} (q_j^e \;\&\; \gamma_j^e \Rightarrow r(\bar{x}^e))$$

which are the correct verification conditions.

(g) Because of (a) - (f) we know that for all PL-programs without while-loops the model generates the verification conditions.
Now suppose there are k basic paths from $q_1,\ldots,q_k$ leading to

maintain p(x)
while b(x) do
S

We want to add the lemmas

q_j & γ_j => $p(\bar{x})$

to τ, where $\bar{x}$ denotes the variable values immediately before the maintain-while statement.
We know by induction hypothesis that if we replace this maintain-while statement by

assert p(x)

the model will generate these verification conditions at that point. But we observe that the first step in the translation of the maintain-while statement is exactly to perform this assertion. Thus the desired effect is achieved.
The only way of entering the statement S is through assuming p(x) and b(x) (which is carried out in the second step of the translation of maintain-while) and we can leave it only by asserting p(x) again.
That, however, is exactly the way of processing

assert p(x) & b(x)
S
assert p(x)

in the verification model.
So we know by induction hypothesis that for all the basic paths ending in S or leading back to the top of the maintain-while statement, the correct verification conditions are added to τ.

Finally we want to start a new basic path by assuming p(x) & b(x) and that is exactly what happens in the model.

References

1. Abdali, S. Kamal: A lambda-calculus model of programming languages Part I & II. Journal of Computer Languages, 1, pp. 287-320, 1976.
2. Abdali, S. Kamal: "CLONE - a combinatory-logical normal form evaluator," User's Manual, Rensselaer Polytechnic Institute, Troy, NY, 1978.
3. Church, A.: The Calculi of Lambda-Conversion. Princeton University Press, NJ, 1941.
4. Floyd, R. W.: Assigning meanings to programs. Math. Aspects of Computer Science. J. T. Schwartz, Ed., Amer. Math. Soc., Providence, R.I., 1967, pp. 19-32.
5. Kaltofen, Erich & Abdali, S. Kamal: An attributed LL(1) compilation of Pascal into the lambda-calculus, Tech. Report, Rensselaer Polytechnic Institute, Troy, NY, June 1981.
6. King, James C.: "A program verifier," Proc IFIP, 1971, pp. 234-249.
7. Manna, Z.: Mathematical Theory of Computation. McGraw Hill, New York, 1974.
8. Morris, James H. Jr. & Wegbreit, Ben: Subgoal induction. Com. ACM 20, 4(April 1977), pp. 209-222.
9. Petznick, G. W.: "Introduction to combinatory logic," in Brainerd, W. S. & Landweber, L. H.: Theory of Computation, John Wiley, New York, 1974.
10. Wijugarden, A. van (Ed.): Report on the algorithmic language ALGOL 68, Numerische Math. 14, 79, 1969.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CS-8104	2. GOVT ACCESSION NO. AD-A103 751	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A LAMBDA-CALCULUS MODEL FOR GENERATING VERIFICATION CONDITIONS		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) S. Kamal Abdali Franz Winkler		8. CONTRACT OR GRANT NUMBER(s) ONR N00014-75-C-1026
9. PERFORMING ORGANIZATION NAME AND ADDRESS Mathematical Sciences Department Rensselaer Polytechnic Institute Troy, New York 12181		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Resident Representative 715 Broadway-5th Floor, N.Y., N.Y. 10003		12. REPORT DATE June 1981 13. NUMBER OF PAGES 29
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Program verification, inductive assertion, verification condition, lambda-calculus

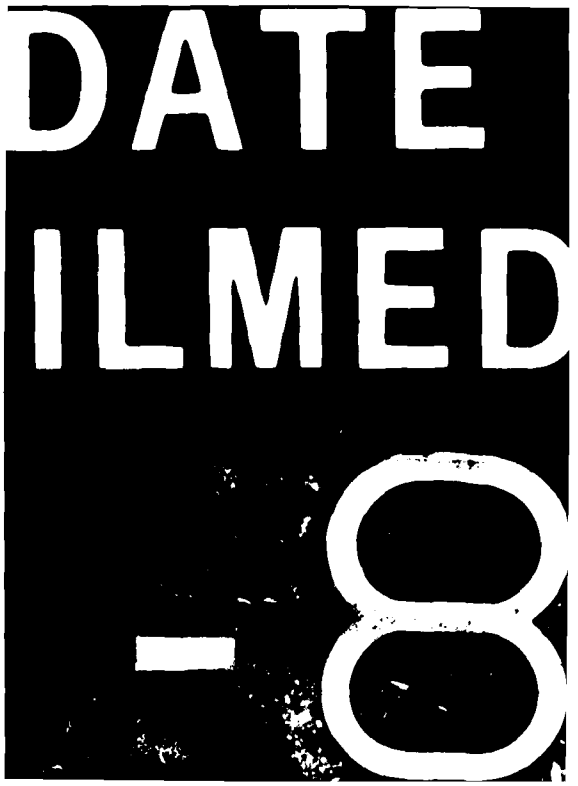
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(see back-side of page)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A lambda-calculus-based method is developed for the automatic generation of verification conditions. A programming language is specified in which inductive assertions associated with a program are incorporated within the body of the program by means of assert and maintain-while statements. This programming language includes the following features: assignments, conditionals, loops, compounds, ALGOL-type block structure. A model is developed which consists of rules to translate each statement in this programming language into the lambda-calculus. The model is such that the lambda-expression translation of any program reduces to a list (tuple) of lambda-expression representations of all verification conditions of the program. A proof of this property is given.


DATE
ILMED
-8